SNS COLLEGE OF ENGINEERING

Coimbatore-107

An Autonomous Institution

Accredited by AICTE and Accredited by NAAC – UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

COURSE NAME: ANALYSIS OF ALGORITHMS

II YEAR/ IV SEMESTER

UNIT –1 INTRODUCTION

Topic

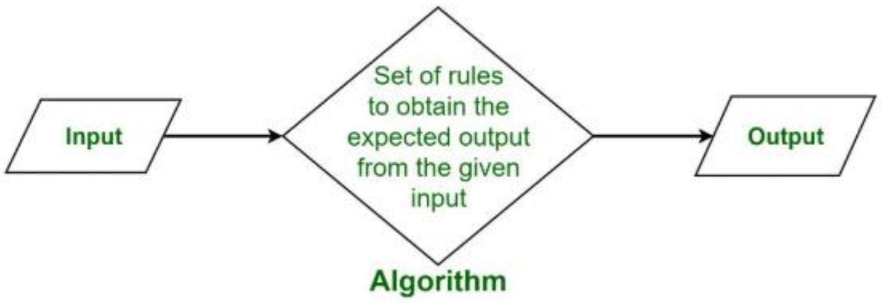
Notion of an Algorithm –Fundamentals of Algorithmic Problem Solving– Important Problem Types – Fundamentals of the Analysis of Algorithm Efficiency – Analysis Framework – Asymptotic Notations and its properties – Mathematical analysis for Recursive and Non-recursive algorithms.

TOPIC1: NOTION OF ALGORITHM

Algorithm:

Algorithm refers to a sequence of finite steps to solve a particular problem. It is collection of instructions (without proper syntax) that provides Output for the given set of Input.

Output can be either actual result or result with some errors for the given problem.


Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Characteristics of Algorithm:

- **Non-Ambiguity:** The algorithm should be unambiguous. Each of its steps should be clear and precise in all aspects and should have no conflict meaning.
- **Well-Defined Inputs and Outputs**: An algorithm has zero or more inputs. Range of inputs must be specified. The algorithm must clearly define output that will be yielded. It should produce at least 1 output.
- **Finiteness:** The algorithm must terminate after performing operation and producing output in finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language,
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out easily. It should possess multiplicity.

Sections of Algorithm:

To write an Algorithm, it should include 2 sections.

1. Algorithm Heading
2. Algorithm Body

Algorithm heading includes name, problem description, input and output.

Algorithm body comprises of Programming constructs and Operators. Datatypes need not to be included. Programming Constructs include if, for, if..else and while statements.

Simple Example:

TO DESIGN AN ALGORITHM TO VERIFY A NUMBER IS EVEN OR ODD.

```
Algorithm evenodd(num)//Algorithm Heading
//Problem Description: To verify whether a number is even or odd
//Input: Get an Input num;
//Output: To check whether a given number is even or odd
START //Algorithm Body
if(num%2==0)
display("Number is Even")
else
display("Number is Odd")
END
```

TOPIC 2-FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

While designing and analysing an algorithm, the following steps are to be followed,

1.Understanding the Problem

•This is the first step in designing of algorithm.

•Read the problem's description carefully to understand the problem

statement completely.

•Ask questions for clarifying the doubts about the problem.

•Identify the problem types and use existing algorithm to find solution.

•Input (instance) to the problem and range of the input get fixed

2.The Decision making:

1. Choice of the Computational Device is done for running different forms of algorithm.

Random-access machine (RAM) executes instructions one after another. Algorithms designed to be executed on such machines are called **sequential algorithms**

→In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called **parallel algorithms.**

2. Choosing between Exact and Approximate Problem Solving:

→An algorithm used to solve the problem exactly and produce correct result is called an **Exact algorithm.**

→If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an **Approximation algorithm**. i.e., produces an →Approximate answer.

3. Choosing Data Structures &Algorithm Design Techniques:

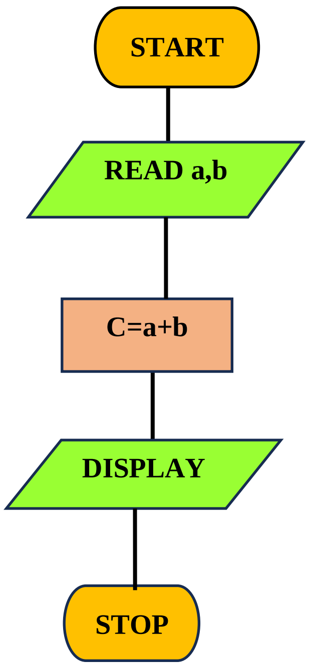
Data Structures is categorized as Linear Data Structure (e.g. Stack, Queue) and Non-Linear Data Structure (e.g. Tree, Graph)

Algorithmic Strategy includes (E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique).

4. Choose Methods of Specifying an Algorithm There are three ways to specify an algorithm. They are: a. Natural language b. Pseudocode c. Flowchart

1. **Natural Language:** We express the Algorithm in the natural English language. It is easy to express but too hard to understand the specification of algorithm from it.
   Example: Addition of two nos:
   Step 1: Read first number a.
   Step 2: Read first number b.
   Step 3: Add two numbers a,b and store result in c.
   Step 4: Display the result:

2. **Flowchart:** We express the Algorithm by making a graphical/pictorial representation of it. It is easier to understand than Natural Language. It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.



3. **Pseudo Code:** Pseudocode is a mixture of a natural language and programming language constructs like if condition, while and for loops. Pseudocode is usually more precise than natural language. But it has no syntax like any of the programming languages. It can't be compiled or interpreted by the computer.

```
Step 1: Read a
Step 2: Read b
Step 3: c= a+,b
Step 4: Display c
```

3. Proving an Algorithm's Correctness Once an algorithm has been specified then its correctness must be proved.

• A common technique for proving correctness is to **use mathematical induction** because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

4. Analyze an Algorithm:

For a standard algorithm to be good, it must be efficient.

Analysis of Algorithm Complexity:

An algorithm is defined as complex based on the amount of Space and Time it consumes.

1.Time complexity of an algorithm refers to the amount of time required by the algorithm to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

The time complexity of an algorithm is determined by following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, arithmetic operations, etc.
- **Variable Time Part:** Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.
  Time complexity of any algorithm P is **f(n) = C + T(I)**, where C is the constant time part and T(I) is the variable part of the algorithm, which depends on the instance characteristic I.

2.Space Complexity of an algorithm refers to the amount of memory required by the algorithm to store the variables and get the result. This can be for inputs, temporary operations, or outputs.
The space complexity of an algorithm is determined by the following 2 components:
Fixed Part: This refers to the space required by the algorithm. For example, input variables, output variables, program size, etc.

Variable Part: This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.

Space complexity **S(n)** of any algorithm P is **S(n) = C + S(I)**, where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Two Forms of Analysis: Analysis of Complexity cane be given by two forms,

1.Mathematical Analysis: calculates basic operation and find recurrence relation. Then it uses any of the 2 methods to find its Time Complexity.

1.Master Theorem

2.Substitution Method

Note:

The Above 2 methods calculate efficiency (Complexity) of Recursive Algorithms only.

2.Empirical Analysis: Empirical Analysis is Solved by Frequency Count Method by

1. Counting the number of loops

2.Iterations made by all the loops and statements within the loops.

Note:

Empirical Analysis and Summation(Mathematical Analysis)can be used to find efficiency(complexity) of Non Recursive algorithm only.

5. Coding an Algorithm:

The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA and it's dependent over the Hardware. Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, replacing expensive operations by cheap ones, selection of programming language should be known to the programmer.

EXAMPLE1 For Simple Statements without using Loops

TO DESIGN AN ALGORITHM FOR ADDING TWO NUMBERS AND FIND ITS TIME COMPLEXITY, SPACE COMPLEXITY AND ITS ORDER.

```
Algorithm sum(a,b)//Algorithm Heading
//Problem Description: To find the sum of two numbers.
//Input: Get 2 inputs a,b
//Output: Added two numbers and provide the result
START //Algorithm Body
a=10
b=5
c=a+b// added two numbers
```

END

i). Time Complexity:

For execution of Simple statements like assignment and arithmetic operations, it takes 1 unit of time for each statement.

For execution of Complex statements such as Looping statements, we use a method called Frequency Count method to calculate its Time Complexity.

Calculated Time Complexity for adding Two numbers:

$$\left.\begin{array}{l} \text{a=10--------------> 1 unit} \\ \text{b=5----------------> 1 unit} \\ \text{c=a+b-------------->1 unit} \end{array}\right\} \quad \textbf{Totally------------> 3 units of time is taken}$$

Time Complexity= f(n)=3

Calculating Order:

Since ‘3’ is constant, degree can be calculated as 1. So, Order= O (1)

ii). Space Complexity:

Space taken for each variable is considered as 1.

Variables Used	**Space Required**
‘a’ stores constant	1
‘b’ stores constant	1
‘c’ stores constant	1
Totally	3

Space Complexity= S(n)=3

Calculating Order:

Since ‘3’ is constant, degree can be calculated as 1. So,

Order= O(1)

EXAMPLE2: FREQUENCY COUNT METHOD used for looping statements/recursion

TO DESIGN AN ALGORITHM FOR ADDING SUM OF ELEMENTS IN AN ARRAY AND FIND ITS TIME COMPLEXITY, SPACE COMPLEXITY AND ITS ORDER.

```
Algorithm sum(A,n)
//Problem Description: To find the sum of elements in an array.
//Input: Get an Array input A[], n as number of elements in Array
//Output: Added elements in an array A[].
START
S=0;
for(i=0;i<n;i++)
{
S=S+A[i]
}
return S
END
```

i). Time Complexity:

For execution of Complex statements such as Looping statements, we use a method called **Frequency Count method** to calculate its Time Complexity.

Calculated Time Complexity for sum of elements in an Array:

No	Statements	Time Taken for each statement
Stmt1	S=0;	**1**
Stmt2	for(i=0;i<n;i++)	**n+1**
Stmt3	S=S+A[i]	**n**
Stmt4	return S	**1**
	Totally	**2n+3 units of time**

Time Complexity= f(n)=2n+3

Explanation:

4 statements are there in algorithm.

1.Stmt1 and Stmt 4 are simple statements. So, time taken is '1' unit of time for each statement.

2.Each loop is executed 'n+1' number of times. So, Stmt 2 is executed 'n+1' times

3. Statements inside the loops will be executed 'n' number of times. So Stmt 3 is executed 'n' times.

// **Note:** Given below part only for understanding

How to calculate as 'n+1' times for loop statements

for(i=0;i<n;i++);

Statements	**Explanation**	**Time taken for each statement**
i=0	Simple statement	1 unit
i<n	1.i=0; and n=2; 0<2;true; loop runs '1' time stmt inside loop run '1' time 'i' incremented to 1 2. i=1; and n=2; 1<2;true; loop runs '1' time stmt inside loop run '1' time 'i' incremented to 2 3. i=2; and n=2; 2<2;false; loop runs '1' time but stmt inside loop does not run. Execution Stops; **Loop runs 3 times; i.e. 'n+1'times** **Statement inside loop runs 2 times; i.e. 'n' times**	n+1 unit

i++	In Above, Stmt inside loop Executed 2 times i.e. 'n' times	'n' unit

//Above part is given only for understanding

Calculating Order:

degree of polynomial is n. So,

Order= O(n)

ii). Space Complexity:

Variables Used	**Space Required**
Array A[] stores 'n' number of Elements	n
'n'=2 is constant	1
's' stores constant	1
'i' stores constant	1
Totally	n + 3

Space Complexity= S(n)=n+3

Calculating Order:

Degree of Polynomial is 'n'

So, Order= O(n)

TOPIC 3-IMPORTANT PROBLEM TYPE

An algorithm should be optimized in terms of time and space. Different types of problems require different types of algorithmic techniques to be solved in the most optimized manner. There are many types of algorithms.

1. Brute Force Algorithm:

- This is the most basic and simplest type of algorithm. It is an intuitive, direct, and straightforward technique of problem-solving in which all the possible ways or all the possible solutions to a given problem are enumerated.

Example: Brute force strategy exploring all the paths to a nearby market to find the minimum shortest path.

2. Recursive Algorithm:

- **Recursion** is technique used in computer science to solve big problems by breaking them into smaller, similar problems. The process in which a function calls itself directly or indirectly is called **recursion** and the corresponding function is called a recursive function.

Eg: Factorial of a Number, Fibonacci Series, Tower of Hanoi, DFS for Graph, etc.

a) Divide and Conquer Algorithm:

- In Divide and Conquer algorithms, the idea is to solve the problem in two sections, the first section divides the problem into subproblems of the same type. The second section is to solve the smaller problems independently and then add the combined result to produce the final answer to the problem.
- Example: Search, Merge Sort, Quick Sort, Strassen's Matrix Multiplication, etc.

b) Dynamic Programming Algorithms:

- It is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization typically reduces time complexities from exponential to polynomial.
- Eg: Fibonacci Numbers, Bellman–Ford Shortest Path and Matrix Chain Multiplication.

c) Greedy Algorithm:

- **Greedy algorithms** are a class of algorithms that make **locally optimal** choices at each step. At every step, we can make a choice that **looks best at the moment,** and we get the optimal solution to the complete problem.

Examples: Knapsack, Dijkstra's algorithm, Kruskal's algorithm, Huffman coding and Prim's Algorithm

d) Backtracking Algorithm:

Backtracking algorithms are like problem-solving strategies that help explore different options to find the best solution. They work by trying out different paths and if one doesn't work, they backtrack and try another until they find the right one. It's like solving a puzzle by testing different pieces until they fit together perfectly.
Eg: the Hamiltonian Cycle, M-Coloring Problem, N Queen Problem, etc.

3. Randomized Algorithm:

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called a Randomized Algorithm

Example:

Quicksort: In Quicksort we use the random number for selecting the pivot.

Karger's algorithm, we randomly pick an edge.

4. Sorting Algorithm:

The sorting algorithm is used to sort data in maybe ascending or descending order**,** according to a comparison operator on the elements.

Example: Bubble sort, insertion sort, merge sort, selection sort, and quick sort

5. Searching Algorithm:

The searching algorithm is the algorithm that is used for searching the specific key in particular sorted or unsorted data. various applications are done in **databases, web search engines**

Example: Binary search or linear search

6. Hashing Algorithm:

Hashing algorithms refers to the process of generating a fixed-size output from an input of variable size using the mathematical formulas known as hash functions. This technique determines an index or location for the storage of an item in a data structure.
Example: Hashing Algorithm used in password verification.

7.Graph Algorithm:

Graph is a collection of vertices and edges. The graph problem involves graph traversal algorithms, shortest path algorithms and topological sorting. Some graph problems are very hard to solve. Example: Travelling Salesman Problem, Graph Coloring Problems.

TOPIC4- Asymptotic Notations and its properties

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

There are mainly three asymptotic notations:

1. Big-O Notation (O-notation)
2. Omega Notation (Ω-notation)
3. Theta Notation (Θ-notation)

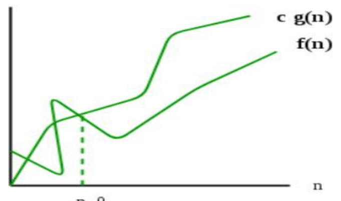
1. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it provides the worst-case complexity of an algorithm. It is the most widely used notation. It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time

Mathematical Representation of Big-O Notation:

If f(n) describes the running time of an algorithm, then f(n) is O(g(n)).if there exist a positive constant C and n0 such that,

$$f(n) \le c * g(n) \text{ for all } n \ge n0$$



f(n) = O(g(n))

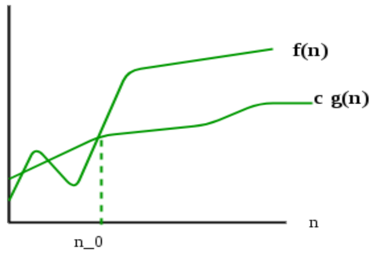
2. Omega Notation (Ω-Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best-case complexity of an algorithm. It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Mathematical Representation of Big-O Notation:

Let g and f be the function from the set of natural numbers. The function f is said to be Ω(g(n)), if there is a constant c > 0 and a natural number n0 such that,

$f(n) \geq c*g(n)$ for all $n \geq n0$



f(n) = Omega(g(n))

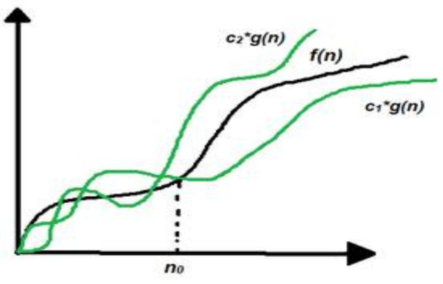
3. Theta Notation (Θ-Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.

Mathematical Representation of Big-O Notation:

Let g and f be the function from the set of natural numbers. The function f is said to be Θ(g(n)), if there are constants c1, c2 > 0 and a natural number n0 such that,

$c1* g(n) \leq f(n) \leq c2 * g(n)$ for all $n \geq n0$



f(n)= Θ(n)

Properties of Asymptotic Notations:

1. General Properties:

If **f(n)** is **O(g(n))** then **a*f(n)** is also **O(g(n))**, where **a** is a constant.

Example:

$f(n)=3n^2+4$ is $O(n^2)$

then, $7*f(n) = 7(3n^2+4) = 21n^2+28$ is also $O(n^2)$. (because highest degree of polynomial is n^2)

Note:

Similarly, this property satisfies both Θ and Ω notation.

If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)), where a is a constant.

If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)), where a is a constant.

2. Transitive Properties:

If **f(n)** is **O(g(n))** and **g(n)** is **O(h(n))** then **f(n) = O(h(n))**.

Example:

Note: Remember example in class (if a=b;b=c; then a=c)

If $f(n) = n$, $g(n) = n^2$ and $h(n)=n^3$ then,

f(n) is O(g(n)) $=O(n^2)$

g(n) is O(h(n)) $= O(n^3)$

then, f(n)is O(h(n)) $=O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n))

If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

3. Reflexive Properties:

If **f(n)** is **given** then,

f(n) is **O(f(n))**.

Example:

$f(n) = n^2$; Order= $O(n^2)$ i.e $O(f(n))$

Similarly, this property satisfies both Θ and Ω notation.

We can say that,

If $f(n)$ is given then $f(n)$ is $\Theta(f(n))$.

If $f(n)$ is given then $f(n)$ is $\Omega(f(n))$.

4. Symmetric Properties:

If **f(n)** is **Θ(g(n))** then,

g(n) is **Θ(f(n))**.

Example:

If(n) = n^2 and $g(n) = n^2$

then, $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

5. Transpose Symmetric Properties:

If **f(n)** is **O(g(n))** (Upper Bound), then,

g(n) is **Ω (f(n))** (Lower Bound)

Example:

If $f(n) = n$, $g(n) = n^2$

then n is $O(n^2)$ and n^2 is $\Omega(n)$

This property only satisfies O and Ω notations.

6. Some More Properties:

1. If f(n) = O(g(n)) (Upper Bound) and f(n) = Ω(g(n)) (Lower Bound), then,

f(n) = Θ(g(n))

note:

$g(n) \leq f(n) \leq g(n)$ [since it swings between upper and lower bound and taken as avg bound]

2. If f(n) = O(g(n)) and d(n)=O(e(n)) then,

$f(n) + d(n) = O(\max(g(n), e(n)))$

Example:

$f(n) = n$ i.e $O(n)$

$d(n) = n^2$ i.e $O(n^2)$

then $f(n) + d(n) = n + n^2$ i.e $O(\max(n,n^2))=O(n^2)$

3. If **$f(n)=O(g(n))$** and **$d(n)=O(e(n))$** then **$f(n) * d(n) = O(g(n) * e(n))$**

Example:

$f(n) = n$ i.e $O(n)$

$d(n) = n^2$ i.e $O(n^2)$

then $f(n) * d(n) = n * n^2 = n^3$ i.e $O(n^3)$

Topic 5-Fundamentals of the Analysis of Algorithm Efficiency

ANALYSIS OF WORST CASE, BEST CASE AND AVERAGE CASE EFFICIENCIES

1. Worst Case Analysis:

- In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.

Example:

- For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search () function compares it with all the elements of arr[] one by one.
- This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

2. Best Case Analysis:

- In the best-case analysis, we calculate the lower bound on the running time of an algorithm. Here Algorithm takes minimum amount of time to execute instructions for specific Input.

Example:

- For linear search, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So, the order of growth of time taken in terms of input size is constant.

3. Average Case Analysis:

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs.

  =All Possible case time/No of Cases

- We must know (or predict) the distribution of cases.

Example:

For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). we sum all the cases and divide the sum by (n+1). We take (n+1) to consider the case when the element is not present.

BASIC ASYMPTOTIC EFFICIENCY CLASSES

Input	Classes of Function	Order represented
1	:Constant.	O (1)
log n..	Logarithmic	O (log n)
n log n	linearithmic	O (n log n)
n^2	Quadratic	O(n^2)
n^3	Cubic	O (n^3).
2^n	Exponential Form	O(2^n)

ORDER OF GROWTH

Measuring the performance of Algorithm in relation with Input Size ‘n’ is called Order of Growth.

Example 1: 4n2 + 3n + 100
After ignoring lower order terms, we get
4n2
After ignoring constants, we get
n2
Hence order of growth is n2

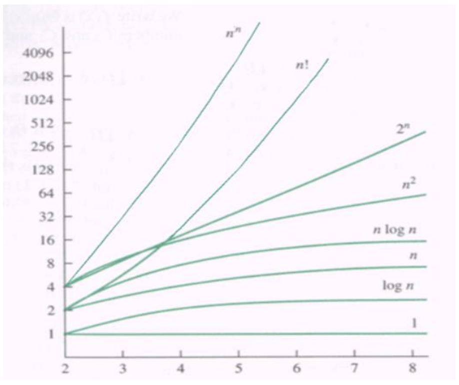
Example 2: 100 n Log n + 3n + 100 Log n + 2
After ignoring lower order terms, we get
100 n Log n
After ignoring constants, we get
n Log n
Hence order of growth is n Log n

Comparing Classes of Functions /orders of growths:

Standard terms for comparison:

c < Log Log n < Log n < n1/3 < n1/2 < n < n Log n < n2 < n2 Log n < n3 < n4 < 2n < nn t

	log n	**n**	**n^2**	**2^n**
n=1	0	1	1	2
n=2	1	2	4	4
n=4	2	4	16	16
n=8	3	8	64	256



TOPIC 6- ANALYSIS FRAMEWORK

1.Measuring Input Size:

Algorithm runs longest time for longer input. Efficiency depends on Input Size that may be Exact or Approximate size.

2.Measuring Running Time:

Refer Notes of Time Complexity with Frequency Count

3.Measuring Space Complexity

4.Measuring Time Complexity

5.Computing Best Case, Worst Case and Average case Efficiencies

6.Order of Growth

NOTE:

/* For topic 3 to 6 Refer previous topics for these notes. */

Topic 7- Mathematical analysis for Nonrecursive algorithms

Mathematical analysis of non-recursive algorithms involves determining time and space complexity using mathematical techniques.

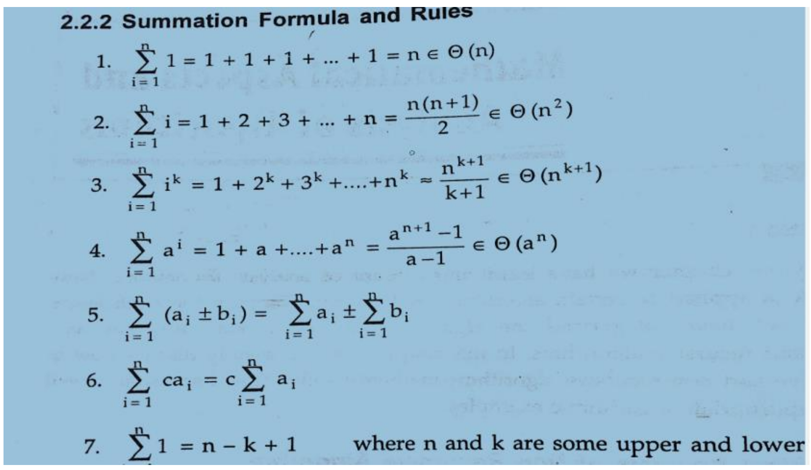
The key steps in analysing involves:

1. Identifying the Basic Operations

- Determine the key operation(s) that contribute most to execution time (e.g., comparisons, additions, swaps).
- Count how many times this operation is executed as a function of input size **n**.

2. Expressing the Algorithm's Time Complexity

- Use a **loop-based analysis** if there are loops.
- Consider how the number of iterations changes with input size.
- Use summation formulas (Mathematical Analysis) and simplify it using standard formulas.



2.2.2 Summation Formula and Rules

1. $\sum_{i=1}^{n} 1 = 1 + 1 + 1 + \dots + 1 = n \in \Theta(n)$
2. $\sum_{i=1}^{n} i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} \in \Theta(n^2)$
3. $\sum_{i=1}^{n} i^k = 1 + 2^k + 3^k + \dots + n^k \approx \frac{n^{k+1}}{k+1} \in \Theta(n^{k+1})$
4. $\sum_{i=1}^{n} a^i = 1 + a + \dots + a^n = \frac{a^{n+1}-1}{a-1} \in \Theta(a^n)$
5. $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$
6. $\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$
7. $\sum^{n} 1 = n - k + 1$ where n and k are some upper and lower

3. Asymptotic Complexity Analysis

- Identify the worst-case, best-case, and average-case complexities.

For non-recursive algorithms, the complexity depends largely on loops and their nesting. By systematically counting iterations and using asymptotic notations, we can derive time complexities efficiently.

Example:

The **factorial problem** involves computing the factorial of a given number **n**, denoted as **n!** which is defined as: **n!=n×(n−1)×(n−2)×⋯×1**

Iterative Factorial (Non-Recursive)Algorithm:

Program	Output execution
	n=5;
int factorial (int n)	1. i=2; result=1;
{	result= 1*2=2; / i incremented by 1
int result = 1;	2.i=3; result=2;
for (int i = 2; i <= n; i++)	result=2*3=6 / i incremented by 1
{	3.i=4; result=6;
result = result * i;	result=6*4=24/ i incremented by 1
}	4. i=5; result=24;
return result;	Result-24*5=120/ i reaches n; stop exec;
}	

Step-by-Step Analysis

1. Identifying the Basic Operations

- The key operation here is multiplication (result = result * i),that takes execution time as O(1).

2. Count the iterations

- Loop runs **from 2 to n**, meaning it iterates **(n - 1)** times. Statement inside loop execute for n times. So **totally (n-1) +n=2n-1.**
- Since constant factors (i.e. 1) is ignored and degree of polynomial is 'n', Order is **O(n)**.

- **Time Complexity: f(n)=O(n)**

3. Asymptotic Complexity Analysis

- **Worst-case complexity:** O(n)
  The loop always executes **(n-1) ≈ n** iterations, making it **linear** in complexity.
- **Best-case complexity:** O(n)
  The loop always runs the same number of times, so best-case is also O(n)
- **Average-case complexity:** O(n)
  Since there is no variation in execution time, the average-case is also **O(n)**.

4. Space Complexity Analysis

- The algorithm uses a single variable (result) to store the output.
- Space complexity is **S(n)=O(1)** (constant space), since no additional memory grows with input size.

Topic 8- Mathematical analysis for Recursive algorithms

Recursive Algorithm:

A recursive algorithm is an algorithm that uses recursion to solve a problem. The idea is to represent a problem in terms of one or more smaller problems, that stops execution at base case.

General Plan for Analysing the Time Efficiency of Recursive Algorithms:

1. Decide on a parameter (or parameters) indicating an input's size.

2. Identify the algorithm's basic operation.

3. Their efficiency is analysed **using recurrence relations**, which describe the function's time complexity in terms of itself. It includes 2 parts,

> **1. Base case:** A recursive program stops at a base condition. There can be more than one base conditions in a recursion.
>
> **2.Recursive case:** Which is a call to the function itself with a smaller version of the problem.

4. Solve the recurrence using either Master theorem or Substitution or Recursion Tree Method.

5.Identify best-case, worst-case, and average-case complexities.

Types of Recurrence Relation:

There are different types of recurrence relation that can be possible in the mathematical world. Some of them are-

1. Linear Recurrence Relation: In case of Linear Recurrence Relation every term is dependent linearly on its previous term. Example of Linear Recurrence Relation can be

T(n) = T(n-1) + T(n-2) + T(n-3)

2. Divide and Conquer Recurrence Relation: It the type of Recurrence Relation which is obtained from Divide and Conquer Algorithm. Example of such recurrence relation can be

T(n) = 3T(n/2) + 9n

3. First Order Recurrence Relation: It is the type of recurrence relation in which every term is dependent on just previous term. Example of this type of recurrence relation can be-

T(n) = T(n-1)2

(4) Higher Order Recurrence Relation- It is the type of recurrence relation where one term is not only dependent on just one previous term but on multiple previous terms. If it will be dependent on two previous term then it will be called to be second order. Similarly, for three previous term its will be called to be of third order and so on. Let us see example of an third order Recurrence relation

T(n) = 2T(n-1)2 + KT(n-2) + T(n-3)

Methods for solving recurrence relations:

- **Substitution Method** (Expanding the recurrence)
- **Master Theorem** (For standard divide-and-conquer recurrences)

1. **Substitution Method:**

Substitution Method is very famous method for solving any recurrences. There are two types of substitution methods-

1. Forward Substitution
2. Backward Substitution

Forward Substitution:

It is called Forward Substitution because here we substitute recurrence of any term into next terms. It uses following steps to find Time using recurrences-

- Pick Recurrence Relation and the given initial Condition
- Put the value from previous recurrence into the next recurrence
- Observe and Guess the pattern and the time
- Prove that the guessed result is correct using mathematical Induction.

Backward Substitution:

It is called Backward Substitution because here we substitute recurrence of any term into previous terms. It uses following steps to find Time using recurrences-

- Take the main recurrence and try to write recurrences of previous terms
- Take just previous recurrence and substitute into main recurrence
- Again take one more previous recurrence and substitute into main recurrence
- Do this process until you reach to the initial condition
- After this substitute the the value from initial condition and get the solution

2. Master Theorem Formula:

- The Master Theorem provides a systematic way of solving recurrence relations of the form:

$$T(n) = aT(n/b) + f(n)$$

The Master Theorem Formula to be applied:

If f(n) ∈ θ (n^d), then,

1.T(n)= θ (n^d), if(a<(b^d))

2. T(n)= θ((n^d) * log n), if(a=(b^d))

3. T(n)= θ ((n^ logb(a)), if(a>(b^d))

Note:

Example for Master Theorem is given in Unit 2

Example: Recursive Factorial (Solved with Substitution Method)

```
Algorithm factorial (int n)
{
   if (n == 0 || n == 1)
       return 1;
else
   return n * factorial (n - 1);
}
```

Step 1: Identify General Recurrence Relation:

$$F(n)=F(n-1) *n;$$

The function calls itself with ‘n-1’ times for ‘n’ factorial.

Step 2: Solve the Recurrence Relation (Solved using Substitution Method)

Obtained Recurrence in our Problem,

$$M(n)=M(n-1) + 1$$

- F(n) becomes M(n)=Basic Operation (Multiplication count)
- F(n-1) becomes M(n-1) =Multiplication required to compute factorial (n-1)
- *n=1= To multiply factorial (n-1) by n

Step3: Use Forward Substitution: (Sub n=1,2,3 in our recurrence relation, we find:)

1. M (1) =M (1-1) + 1

   =M (0) +1

=0+1=1(Therefore M (1) =1)

M (2) =M (2-1) + 1

=M (1) +1

=1+1=2(Therefore M (2) =2)

2. M (3) =M (3-1) + 1

=M (2) +1

=2+1=3(Therefore M (3) =3)

Step3: Use Backward Substitution: (Sub n=(n-1), (n-2), (n-3) in our recurrence relation, we find:)

1.M(n-1) =M(n-1-1) +1+1(Multiply count by 'n' each time)

=M(n-2) +2

2.M(n-2) =M(n-2-1) +1+2(Previous Count)

= M(n-3) +3

2.M(n-3) =M(n-3-1) +1+3(Previous Count)

= M(n-4) +4

From Above Substitution, we get a formula.

M(n)=M(n-i) + i

Step 4: Proving Correctness using Mathematical Induction:

1.To Prove: M(n)=n by Mathematical Induction

2.Basis: Put n=0;

M(n)=M(0)=0=n (since n and 0 are equal)

3.Induction: Assume M(n-1)=n-1 ,then

M(n)=M(n-1)+1

=(n-1)+1

=(n-0)=n

Hence Proved, M(n)=n;

Step 5: Determine Asymptotic Complexity

- **Time Complexity:** since T (1) =O (1);

  we get **T(n)=O(n)**

- **Space Complexity**

  S(n) =O(n) (due to recursive calls on the call stack)